[SYSTEM-DESIGN]

Designing Data Intensive Applications – Chapter 3: Storage and Retrieval

Posted by CHARLES *on* 2020-04-02

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

(If you keep things tidily ordered, you're just too lazy to go searching.) —German proverb

- DB needs to do two things:
  - when you give it some data, it should store the data;
  - when you ask it again later, it should give the data back to you.
- How we can store the data that we're given, and how we can find it again when we're asked for it.
  - There is a big difference between storage engines that are optimized for **transactional** workloads and those that are optimized for **analytics**.
- **Two Type of DB Engine**
  - **log-structured** storage engines: (Log: an append-only sequence of records.)
    - many databases internally use a log, which is an append-only data file;
  - **page-oriented** storage engines such as B-trees(more popular)**:**

Data Structures That Power Your Database:

- **Log-Structured** need an Index to speed up look up time: An index is an additional structure that is derived from the primary data;
  - Maintaining additional structures incurs overhead, especially on writes, because it need updated every time on write;
  - Application developer or database administrator to choose indexes manually.
- **Segment Files:** break the log-file into segment
  - **Merging/Compaction:** means throwing away duplicate keys in the log, and keeping only the most recent update for each key.
- **Hash Indexes**: basically use a HashMap(HashTable) to store the index in memory for speed up the lookup time:

    - We require that the sequence of key-value pairs is sorted by key;
    - We also require that each key only appears once within each merged segment file;
    - **Advantages**:
        - Merging segments is simple and efficient, even if the files are bigger than the available memory(kind like mergesort);
        - no longer need to keep an index of all the keys in memory;
        - Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk.
    - **Constructing and maintaining SSTables**: e.g. LevelDB, RocksDB,Cassandra and HBase
        - How to keep it sorted: in-memory tree is sometimes called a **memtable**, e.g. red-black tree or AVL tree;
        - When memtable gets bigger than some threshold, write it to disk;
        - serve a read request: start from memtable → most recent segment → other segments;
- **Making an LSM-tree out of SSTables**:
    - LSM-tree: **Log-Structured Merge-Tree** (or LSM-Tree)
    - Storage engines that are based on this principle of **merging** and **compacting** sorted files are often called LSM storage engines;
    - **Lucene**, an indexing engine for full-text search used by **Elasticsearch** and **Solr**.
- **Performance optimizations**:
    - **Bloom filters**: a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does/may not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.
    - **Size-tiered**: newer and smaller SSTables are successively merged into older and larger SSTables. (e.g.HBase, Cassandra)
    - **leveled compaction**: the key range is split up into smaller SSTables and older data is moved into separate “levels,” which allows the compaction to proceed more incrementally and use less disk space. (e.g. LevelDB and RocksDB, Cassandra)
- **B-Trees(most popular)**:
    - They remain the standard index implementation in almost all relational databases, and many non-relational databases use them too;
    - Like SSTables, B-trees keep key-value pairs sorted by key, but B-trees have a very different design philosophy.
    - B-trees break the database down into **fixed-size blocks or pages**, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time. (Corresponding to Hardware structure, HDD)
    - The number of references to child pages in one page of the B-tree is called the **branching factor**. Growing a B-tree by splitting a page

- **Making B-trees reliable**:
  - Common for B-tree implementations to include an additional data structure on disk: a write-ahead log (**WAL**, also known as a redo log).
    - This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself.
    - protecting the tree's data structures with latches (lightweight locks)
  - **B-tree optimizations**:
    - Instead of overwriting pages and maintaining a WAL for crash recovery, some databases (like LMDB) use a copy-on-write scheme;
    - We can save space in pages by not storing the entire key, but abbreviating it.
- **Comparing B-Trees and LSM-Trees** :
  - **LSM-trees** are typically faster for writes;
    - **Pros** of LSM-trees:
      - sometimes have **lower write amplification**
      - LSM-trees can be **compressed** better
      - they have **lower storage overheads**, especially when using leveled compaction.
    - **Write Amplification**: Rewrite data multiple times due to repeated compaction and merging of SSTables.
    - **Cons** of LSM-trees:
      - The compaction process can sometimes interfere with the performance of ongoing reads and writes.
      - the bigger the database gets, the more disk bandwidth is required for compaction.
      - If write throughput is high and compaction is not configured carefully, it can happen that compaction cannot keep up with the rate of incoming writes.
  - **B-trees** are thought to be faster for reads;
    - This aspect makes B-trees attractive in databases that want to offer strong transactional semantics: in many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree.
- **Other Indexing Structures**:
  - **key-value indexes**, which are like a primary key index in the relational model.
  - **Secondary indexes**, which are often crucial for performing joins efficiently.
    - Either way, both B-trees and log-structured indexes can be used as secondary indexes.
  - **Storing values within the index**:
    - The Value: it could be the actual row (document, vertex) in question, or it could be a reference to the row stored elsewhere(Heap file).
      - Each index just references a location in the heap file, and the actual data is kept in one place.(avoid duplication when multiple secondary indexes)

    - a clustered index (storing all row data within the index)
    - a nonclustered index (storing only references to the data within the index), which stores some of a table's columns within the index.
  - **Multi-column indexes**:
    - **concatenated index**: which simply combines several fields into one key by appending one column to another.
    - **Multi-dimensional indexes** are a more general way of querying several columns at once, which is particularly important for geospatial data
  - **Full-text search and fuzzy indexes**:
    - In Lucene, the in-memory index is a finite state automaton over the characters in the keys, similar to a trie; (C: aka. reverse index)
      - Levenshtein automaton, which supports efficient search for words within a given edit distance(add/remove/update)
- **Keeping everything in memory**: (e.g. Memcached, VoltDB, MemSQL, and Oracle TimesTen, RAMCloud,Redis and Couchbase)
  - Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory,potentially distributed across several machines. This has led to the development of **in-memory databases**.
  - **Persistence**: special hardware (such as battery-powered RAM), by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.
  - They can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk.
  - Providing data models that are difficult to implement with disk-based indexes. (e.g. Redis with Priority-Queue and Sets)

Transaction Processing or Analytics

- **ACID (Atomicity, Consistency, Isolation, and Durability)** properties.
- online transaction processing (**OLTP**) vs. online analytical processing (**OLAP**)

Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

- Databases started being increasingly used for data analytics, which has very different access patterns.
  - These queries are often written by business analysts, and feed into reports that help the management of a company make better decisions (business intelligence **BI**)
- **Data Warehousing**: is a separate database that analysts can query to their hearts' content, without affecting OLTP operations. contains a **read-only copy** of the data in all the various OLTP systems in the company.

- **The divergence between OLTP databases and data warehouses**:
  - Many database vendors now focus on supporting either transaction processing or analytics workloads, but not both. (e.g. Teradata, Vertica, SAP HANA, and ParAccel(RedShift from AWS), SQL-on-Hadoop, Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill, Google Dremel)

Stars and Snowflakes: Schemas for Analytics

- **Star schema** (also known as dimensional modeling): The name "star schema" comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star.
  - At the center of the schema is a so-called **fact table**;
    - fact tables often have over 100 columns, sometimes several hundred;
  - Some of the columns in the fact table are attributes, e.g. price at which the product was sold and the cost of buying it from the supplier;
  - Other columns in the fact table are foreign key references to other tables, called **dimension tables**.
    - The dimensions represent the **who, what, where, when, how, and why** of the event.
- **Snowflake Schema**: where dimensions are further broken down into subdimensions.
- **Column-Oriented Storage**:
  - **Row-Oriented fashion**: all the values from one row of a table are stored next to each other.
  - The idea behind **Column-Oriented storage**: don't store all the values from one row together, but store all the values from each column together instead.
    - If each column is stored in a separate file, a query only needs to read and parse those columns that are used in that query, which can save a lot of work.
  - The column-oriented storage layout relies on each column file containing the rows in the **same order**.
  - **Column Compression**: we can further reduce the demands on disk throughput by compressing data(e.g. bitmap encoding)
    - We can now take a column with n distinct values and turn it into n separate bitmaps: one bitmap for each distinct value, with one bit for each row. The bit is 1 if the row has that value, and 0 if not.
    - If n is very small (for example, a country column may have approximately 200 distinct values), those bitmaps can be stored with one bit per row.
    - if n is bigger, there will be a lot of zeros in most of the bitmaps (we say that they are sparse), the bitmaps can additionally be run-length encoded.
  - **Memory bandwidth and vectorized processing**:

    - The data needs to be sorted an entire row at a time, even though it is stored by column.
    - A second column can determine the sort order of any rows that have the same value in the first column.
    - It can help with compression of columns.
    - **Several different sort orders**:
        - Different queries benefit from different sort orders, so why not store the same data sorted in several different ways.
        - Having multiple sort orders in a column-oriented store is a bit similar to having multiple secondary indexes in a row-oriented store.
  - **Writing to Column-Oriented Storage**:
    - Column-oriented storage, compression, and sorting all help to make those read queries faster.
    - However, they have the downside of making writing more difficult.
        - One solution: LSM-trees. All writes first go to an in-memory store, where they are added to a sorted structure and prepared for writing to disk. (e.g. Vertica)
- **Aggregation**: **Data Cubes** and **Materialized Views**;
  - **Aggregate functions**: such as COUNT, SUM, AVG, MIN, or MAX in SQL.
  - **materialized view**: In a relational data model, it is often defined like a standard (virtual) view: a table-like object whose contents are the results of some query. (Not often in OLTP)
  - Special case of Materialized View: data cube or OLAP cube. It is a grid of aggregates grouped by different dimensions.
    - **The advantage of a materialized data cube** is that certain queries become very fast because they have effectively been precomputed.
    - **The disadvantage is that a data cube** doesn't have the same flexibility as querying the raw data.
    - use aggregates such as data cubes only as a performance boost for certain queries.

Summary

- Storage engine two broad categories:
  - Optimized for **transaction** processing (OLTP)
    - **OLTP** systems are typically user-facing, which means that they may see a huge volume of requests. In order to handle the load, applications usually only touch a small number of records in each query. The application requests records using some kind of key, and the storage engine uses an index to find the data for the requested key. Disk seek time is often the bottleneck here.
  - Optimized for **analytics** (OLAP).
    - Data warehouses and similar analytic systems are less well known, because they are primarily used by business analysts, not by end users. They handle

and column-oriented storage is an increasingly popular solution for this kind of workload.
    - **Indexes are much less relevant**. Instead it becomes important to encode data very compactly, to minimize the amount of data that the query needs to read from disk. (column-oriented storage helps achieve this goal)
- On the OLTP side, we saw storage engines from two main schools of thought:
    - The **log-structured** school, which only permits appending to files and deleting obsolete files, but never updates a file that has been written.
        - Bitcask, SSTables, LSM-trees, LevelDB, Cassandra, HBase, Lucene, and others belong to this group.
    - The **update-in-place** school, which treats the disk as a set of fixed-size pages that can be overwritten.
        - B-trees are the biggest example of this philosophy, being used in all major relational databases and also many non-relational ones.
    - Their **key idea** is that they systematically turn random-access writes into sequential writes on disk, which enables higher write throughput due to the performance characteristics of hard drives and SSDs.

<Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems>

PREVIOUS POST
Happy #chinesenewyear #yearofthepig #piggy

NEXT POST
Designing Data-Intensive Applications – Chapter 1: Reliable, Scalable, and Maintainable Applications

Leave a Reply

Enter your comment here...

This site uses Akismet to reduce spam. Learn how your comment data is processed.